# CHAPTER 3 Privacy and Security

## 3.1 Introduction

This chapter addresses privacy and security concerns in the architecture we described in Chapter 2. It assumes knowledge of the contents of that chapter, but not necessarily in-depth knowledge of modern cryptography or computer security.

We shall describe:

| | |
|---|---|
| • The nature of the problem, including: | Section 3.2 |
| • The threat model, such as what attacks we expect, and the difference between passive and active attackers | Section 3.2.1 |
| • A discussion of how private we are trying to be | Section 3.2.2 |
| • Some desiderata for security design in general, and how our architecture makes use of them | Section 3.2.3 |
| • The problems we are *not* attempting to solve | Section 3.2.4 |
| • Some useful cryptographic techniques, including: | Section 3.3 |
| • Symmetric encryption | Section 3.3.1 |
| • Public-key encryption | Section 3.3.2 |
| • Cryptographic hashes | Section 3.3.3 |
| • Some issues in key distribution | Section 3.3.4 |
| • Some of the solutions we employ, including: | Section 3.4 |
| • How anonymity and pseudonymity help | Section 3.4.1 |
| • Various techniques against passive attackers | Section 3.4.2 |
| • Various techniques against active attackers | Section 3.4.3 |
| • Issues involved in protecting the distribution | Section 3.4.4 |
| • Selected additional topics which tie up some loose ends | Section 3.5 |

## 3.2 The problem

This section discusses the types of attacks the architecture is likely to see, as well as the problems we are *not* trying to solve.

### *3.2.1 The threat model: what attacks may we expect?*

Given the architecture described in the previous chapter, there are a wide variety of potential attacks which may be mounted by malicious or curious third parties. They generally break down into *passive* attacks, in which communications are merely monitored, and *active* attacks, in which communications or the underlying agents themselves are subverted, via deletion, modification, or addition of data to the network.

*Packet sniffing*

**Passive attacks**. The most obvious attack is simple monitoring of packet data; such an attack is often accomplished with a *packet sniffer*, which simply records all packets transmitted between any number of sources. If such data includes users' mail messages or files, then two agents which are trading this information back and forth will leak information to an *eavesdropper.*

*Traffic analysis*

Even if the actual communications between agents are perfectly encrypted, however, passive attacks can still be quite powerful. The easiest such attack, in the face of encrypted communications, is *traffic analysis*, in which the eavesdropper monitors the *pattern* of packet exchange between agents, even if the actual *contents* of the packets are a mystery. This can be surprisingly effective: It was traffic analysis that alerted a pizza delivery service local to the Pentagon—and thus the media—when the United States was preparing a military action at the beginning of the Gulf War; when late-night deliveries of pizza suddenly jumped, it became obvious that something was up [181]. (Even though [179] points out that press coverage of the pizza effect tends to quote unnamed sources, a very small number of individuals with personal stakes—such as a Domino's manager in the area—and other press reports, the continuing press coverage [151] and even military recommendations [174] surrounding such effects make it clear that this threat is taken seriously.)

*Spoofing and replays*

**Active attacks.** Active attacks involve disrupting the communications paths between agents, or attacking the underlying infrastructure. The most common such attack is a *spoofing* attack, in which one agent impersonates another, or some outside attacker injects packets into the communication system to simulate such an outcome. Often, spoofing is accomplished via a *replay* attack, in which prior communications between two agents are simply repeated by the outsider. Even if the plaintext of the encrypted contents of the communication are not known, such attacks can succeed so long as duplicate communications are allowed and the attacker can deduce the effect of such a repeat. For instance, if it is noticed that a cash-dispensing machine will always dispense money if a particular (encrypted) packet goes by, a simple replay can spoof the machine into disgorging additional cash.

*Subverted agents*

More sophisticated attacks are certainly possible. Individual running agents might be subverted by a third party, such that they are no longer trustworthy. Such a subverted agent might use encryption keys which are known to the interloper, for example. Alternately, the attacker might create his or own own agent, which looks like a genuine agent to the rest of the network, but pretends to have characteristics which match *everything*—in Yenta, for example, such an agent might then be used to troll for people interested in particular topics, and presumably also would be modified to disgorge anything interesting to its creator.

*Subverted distribution*

Finally, the actual distributed agent might be modified by a determined attacker at the source itself—say, by subtly introducing a trojan horse into the application at its distribution point(s), either by modifying its source code, or by modifying any precompiled binaries which are being distributed. This is essentially a more-distributed and more-damaging version of the subverted-agent attack above. As an example, consider all the Web pages currently extant which proclaim, "These pages are best viewed with Netscape *x.y*. Download a copy!" Now imagine what would happen if the link pointed to a carefully-modified version of Netscape that always supplied the *same* session key, known to the interloper: the result would be that anyone who took the bait would be running a version of Netscape with no security whatsoever, hence leaving themselves vulnerable to, e.g., a sniffing attack on their credit card number.

Consider the degrees of anonymity offered by the chart below:

**3.2.2 How private is private?**

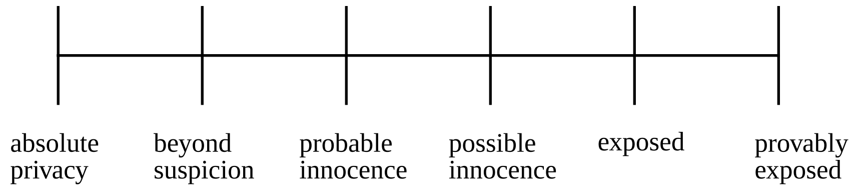

**Figure 3: Degrees of anonymity**

As defined in [141], the extremes of this chart range from *absolute privacy*, where the attacker cannot perceive the presence of communication, to *provably exposed*, where the attacker can prove the sender, receiver, or their relationship to others. The discussion advanced in [141] is oriented more towards the perception of communication *at all*, whereas we are concerned with the *contents* of that communication as well, but the spectrum of possibilities is nonetheless useful. They define the rest of the chart as follows:

- A sender is *beyond suspicion* if, though the attacker can see evidence of a sent message, the sender appears no more likely to be the originator of that message than any other potential sender in the system.
- A sender is *probably innocent* if, from the attacker's point of view, the sender appears no more likely to be the originator than to not be the originator. This is weaker than beyond suspicion in that the attacker may have reason to expect that the sender is more likely to be responsible than any other potential sender, but it still appears at least as likely that the sender is not responsible.
- A sender is *possibly innocent* if, from the attacker's point of view, there is a nontrivial probability that the real sender is someone else. While weaker than the above, it may prevent attackers from *acting* on their suspicions.
- Finally, a sender is *exposed* if an attacker can see information which unambiguously identifies the sender. *This is the default for almost all communications protocols on the Internet*—most such protocols are cleartext, and make no attempt to hide the addresses of senders or receivers. This is weaker than being *provably exposed*, however, since it is generally the identity of the *computer* that is revealed, rather than some nonrepudiable *user* identity.

The architecture discussed here, for the most part, attempts to ensure either *possible innocence* or *probable innocence*; we shall differentiate where useful. In addition, certain parts of the architecture may make it possible for the user to be *beyond suspicion* to a *local* eavesdropper—someone who can monitor some, but not all, communication links in the system.

**3.2.3 Security design desiderata**

The security architecture presented here is cognizant of several principles which are well-known in the security and cryptographic communities. This section discusses several of them, and demonstrates how they have motivated various decisions taken in the design.

*Design must be open—the importance of open source*

*Security through obscurity of design does not work*. This means that any design which depends upon secrecy of the design is guaranteed to fail, since secrets have a way of getting out. Since this architecture is designed to be run by a large number of individuals all across the Internet, its binaries must be public, hence security through obscurity would be untenable anyway in the face of disassemblers and reverse-engineering. (In fact, the source code of Yenta, the sample application, is also public, which should increase confidence in the resulting system; see the discussion of Yvette in Section 3.4.4.)

*Protect the keys*

*Keys are the important entity to protect.* In good cryptographic algorithms, it is the keys that are the important data. Since keys are usually a small number of bits—hundreds or perhaps thousands at most. Because new keys are often trivial to generate, protecting keys is much easier than protecting algorithms. Unfortunately, however, *key management*—keeping track of keys and keeping them from being accidentally disclosed—is often the hardest and weakest point of a cryptosystem [6][15][24][34][69]. Our architecture has a variety of keys and manages them carefully.

*Use existing crypto*

*Good cryptography is hard to design and hard to verify.* Most brand-new cryptographic systems turn out to have serious flaws. Only when a system has been carefully inspected by a number of people is it reasonable to trust it. This is another reason why security through obscurity is a bad idea. We depend on well-established algorithms and protocols for our fundamental security, since they have been carefully scrutinized.

*Whole-system design*

*Security is a function of the entire system, not individual pieces.* This means that even good cryptography and system design is worthless if it can be compromised by bribing or threatening someone. Part of the reason for the decentralized nature of this architecture is to avoid having a single point of compromise, as detailed in Chapter 1.

*Poor design is dangerous too*

*Malevolence and poor design are sometimes indistinguishable.* Many system failures that look like the result of malevolence are instead the result of the interaction of an accident and some unfortunate element of the design. For example, the entire ARPAnet failed one Sunday morning in 1973 due to a double-bit error in a single IMP [121]. A similarly disastrous outcome from a simple, single error is aptly described by this quote: “The whole thing was an accident. No saboteur could have been so wildly optimistic as to think he could destroy an airplane this way,” which described how an aircraft was demolished on a friendly airfield during World War II when someone ingeniously circumvented safety measures and inadvertently connected a mislabelled hydrogen cylinder to the plane’s oxygen system [137].

*Minimize collected information*

*If you don’t want to be subpoenaed for it, don’t collect it.* As we mentioned in Chapter 1, Federal Express, a delivery service in the United States, receives (and hence is compelled to respond to) several hundred subpoenas a day for its shipping records [178]. The safest way to protect private data collected from others from such disclosure—not to mention the hassle of responding to a stream of subpoenas—is never to collect it in the first place. Both the lending records of most libraries, and the logfiles of MIT’s primary mailers—which are guaranteed to be thrown away irretrievably when three days old [153]—adhere to this rule. This also motivates our decentralized design: any central point is a subpoena target.

*We can’t be perfect*

*Security is a spectrum, not an absolute*. A computer can often be made perfectly secure by unplugging it—not to mention vaporizing its disks—and their backups. However, this is a high price to pay. Tradeoffs between security and functionality or performance are often necessary. It is also true that new attacks are constantly being invented; hence, while this research aims at a more-secure implementation than that which is possible without attending to these issues at all, we can never claim to be completely secure. We therefore aim for security that is good enough, and to *do not harm*—such that user privacy is protected as well or nearly as well as it would be if the application was not running. We cannot hope for better—doing better would imply that our application somehow magically improves the security of other, unrelated applications—and may have to make *some* tradeoffs that nonetheless lead to a little bit of insecurity for a large benefit.

## 3.2.4 Problems not addressed

There are a number of problems which are *not* addressed in the security architecture presented here. The problems we are not addressing influence where we will and will not accept design compromises.

*No mobile code*

For instance, since each agent runs on a user's individual workstation, and each agent is not itself a mobile agent per se [29][35][70][170][183], we do not have the problem of executing arbitrary chunks of possibly-untrusted code on the user's local workstation.

*No Byzantine failures*

Further, it is assumed that, while *some* agents may have been deliberately compromised, the vast majority of them have not. This mostly frees us from having to worry about the problems of *Byzantine failure* [53][131] in the system design, wherein a *large* portion of the participants are either malfunctioning or actively malicious.

We also assume, as in the Byzantine case, that not *every* other agent any *particular* agent communicates with is compromised. If this were not true, certain parts of the algorithm would be vulnerable to a *ubiquitous* form of the *man-in-the-middle* attack, wherein an interloper pretends to be A while talking to B, and B while talking to A, with neither of them the wiser. (Weaker forms of this, wherein there are only a *few* agents doing this, have reasonable solutions. In general, when dealing with Byzantine failures, the amount of work to cope with increasing numbers of hostile peers goes up quite rapidly—exponentially in many cases. This means that dealing with a small number of miscreants is feasible, whereas the situation where most peers are untrustworthy becomes very difficult.)

*Trusted path to binaries*

The architecture provides no protection for the user if his or her copy of the application has been compromised. It is generally trivial for a sophisticated attacker to compromise a binary—for example, by substituting NFS packets on the wire as the application is loaded from the fileserver. We cannot be of any help in this case; a user without a trusted path to his or her binaries is already at the mercy of any good attacker, regardless of the application being run.

*Cracked root*

Along the same lines, a user who runs the application on untrusted hardware cannot expect that it can never be compromised—this is analogous to not having a trusted path to one's binaries, since an attacker who has compromised the computer on which the application is being run can by definition either read or alter data in the running binary. Consider the example of Yenta, which runs as a *daemon* and remains resident in memory indefinitely. The user's secret key, which is the basis of his or her identity, must similarly remain in memory for long periods of time. If this were not the case, then the user would have to constantly type his or her passphrase for *every* operation which required an identity check, of which there are many. But this also means that any attacker who has root access to the user's workstation, for example, can read this key out of the process address space. Hence, if the user's workstation has poor security in general, then Yenta's ability to keep the user's secrets from the attacker will be no better.

*Poor passphrases*

While this architecture tries to minimize the number of places where users can inadvertently compromise their own security, some user responsibility is nonetheless expected. For example, the agent must store its permanent state somewhere. If this data is to be private, it must be protected. Absent hardware solutions, the most reasonable solution to this protection is to encrypt it with a passphrase—but nothing can help us if the user chooses a poor passphrase, such as one that is too short or is easily guessed.

*Government agencies or rubber-hose cryptanalysis*

Similarly, this architecture is no protection against the resources of a government agency, or some similarly-equipped adversary. Such an adversary has no reason to attempt a subtle compromise of the distribution, the protocols, or the cryptography. It may instead physically bug the user's premises, compromise his hardware, or use *rubber-hose cryptography*—coercing the user's key(s) via implied or explicit threat of physical force. A possible solution to coercion is the use of *deniable filesystems* [22], but this is beyond the scope of the research presented here.

*No denial-of-service*

In addition, we do not explicitly deal with *denial-of-service* attacks, which are extremely difficult for any distributed system to address. Such attacks amount to, for example, dropping every packet between two agents which are trying to communicate—this attack looks like the network has been partitioned to the agents involved, and there is little defense.

*No international export*

Finally, we have the problem of our use of strong cryptography to protect users' privacy. The United States government currently regulates such cryptographic software as a munition, under *EAR*, the Export Administration Regulations [50]—formerly *ITAR*, the International Treaty On Arms Regulations [87]. This means, for example, that the cryptographic portions of Yenta's software are currently unavailable outside the US unless added back in elsewhere. Solving the limitations of EAR/ITAR is not explicitly addressed here—except to demonstrate how such governmental policies work against the sovereign rights of its citizens, as we detail in Chapter 1.

## 3.3 Cryptographic techniques

This section introduces some useful cryptographic techniques that will be used later. The techniques we discuss are used as *black boxes*, without proof that they properly implement the functionality described for the box and without the mathematical background which underlies them; those who wish to check these assertions may examine the citations where appropriate. In particular, for a much more complete introduction that includes an excellent survey of the field, see [155].

### *3.3.1 Symmetric encryption*

One of the most straightforward cryptographic techniques uses *symmetric keys*. Algorithms such as IDEA ([155] pp. 319-324) work this way. Given a 128-bit key, the algorithm takes *plaintext* and converts it to *ciphertext*. Given the same key, it also converts ciphertext back into plaintext. Expressed mathematically, we can say that C=K(P) [the ciphertext C is computed from the plaintext P via a function of the key K], and similarly P=K(C) [the reverse also works].

IDEA is probably very secure. The problem comes in *distributing the keys*: we cannot just transmit the keys before the encrypted message—after all, the channel is deemed insecure or we wouldn't need encryption in the first place—hence users must first meet *out-of-band*, e.g., not using the insecure channel, to exchange keys. This is infeasible for a large variety of applications.

### *3.3.2 Public-key encryption*

A better approach uses a *public-key cryptosystem* [PKC], such as RSA ([155] pp. 466-473) or the many other variants of this technology. In a public key system, each user has *two* keys: a *public* key and a *private* key, which must be generated together—neither is useful without the other. As its name implies, each user's public key really is public—it can be published in the newspaper. The private key, on the other hand, is *never* shared, not even with someone the user wishes to communicate with.

*Confidentiality*

User A encrypts a message to B by computing $C=K_{PB}(P)$, e.g., a function involving B's *public* key. To decrypt, B computes $P=K_{SB}(C)$, e.g., B's *private* key. Note that, once encrypted, A *cannot* decrypt the resulting message, using any key A has access to—the encryption acts *one-way* if A does not have B's private key—and she shouldn't! [One important detail: since PKC's are usually slow, one usually creates a brand-new *session key*, transmits *that* using PKC, then uses the session key with a symmetric cipher such as IDEA or triple-DES to transmit the actual message. In addition, PKC's may sometimes leak bits if used to encrypt large amounts of data; encrypting only keys can avoid this problem.]

*Authenticity*

This scheme provides not only *confidentiality*—third parties cannot read the messages—but also *authenticity*—B can prove that A sent the message. How does this work? Before A sends a message, she first *signs* the message by encrypting it (really a *cryptographic hash* of the message—see below) with *her own private key*. In other words, A computes $P_{signed}= K_{SA}(P)$. Then, A *encrypts* the message to B, computing $C=K_{PB}(P_{signed})$. B, upon receiving the message, computes $P_{signed}=K_{SB}(C)$, which recovers the plaintext, and can then *verify* A's signature by computing $P=K_{PA}(P_{signed})$. B can do this, because he is using A's *public* key to make the computation; on the other hand, for this to have worked at all, A must have sent it, because only her *private* key could have signed the message such that her public key worked to check it. Only if someone had cracked or stolen A's private key could the signature have been fraudulently created.

### 3.3.3 Cryptographic hashes

It is often the case that one merely wishes to know whether some message has been tampered with. One obvious solution is to transmit the message *out of band*—via some channel which is not the same as the channel originally used to transmit the message. But this begs the question of how *that* channel is secured, and can be very inconvenient to implement in any case.

An easy way to avoid out-of-band transmission is via a cryptographic hash, such as MD5 ([155], pp. 436-441) or the Secure Hash Algorithm (SHA, [155], pp. 442-445). These hash functions compute a short (128-bit or 160-bit, respectively) message digest of an unlimited-length original message. These functions have the unusual property that changing any single bit of the original message changes, on average, half of the bits of the digest. Further, they function in a one-way fashion—it is infeasible, given a digest, to compute a message which, when hashed, would yield the given digest.

On the other hand, anyone can compute the hash of a message, since the algorithm is public and uses no keys. This means that it is computationally easy to verify that a particular message does, in fact, hash to a particular value, even though it is infeasible to *find* a message which produces some particular hash.

*Digital signatures*

Since such hashes are compact yet give an unambiguous indication of whether the original message has been altered, they are often used to implement *digital signatures* such as in the RSA scheme above—what is signed is not the actual cleartext message, but a hash of it. This also improves the speed of signing (since signing a 128- or 160-bit hash is much faster than signing a long message), and the actual security of the cipher as well (because RSA is vulnerable to a *chosen-plaintext* attack; see [155], p. 471).

### 3.3.4 Key distribution

One of the hardest problems of most cryptosystems, even public-key systems, is correctly *distributing* and *managing* keys. In a public-key system, the obvious attacks—compromise of the actual private key—are often relatively easy to guard against: keep the private key in memory as little as possible, encrypt it on disk using DES with a passphrase typed in by the user to unlock it [187], and keep it offline on a floppy if possible.

But consider this: Alice wishes to send a message to Bob. She looks up Bob's public key, but interloper Mallot intercedes and supplies *his own* public key. Alice has no way of knowing that Mallot has done so, but the result of her encryption is a message that *only* Mallot, and not Bob, can read! Even if one demands that Alice and Bob have a round-trip conversation to prove that they can communicate, Mallot could be playing man-in-the-middle, simultaneously decrypting and re-encrypting in both directions as appropriate.

*Webs of trust*

To solve this problem, systems such as Privacy Enhanced Mail [92] use a centralized, tree-structured key registry, which is inconsistent with our decentralized, no-hierarchy architecture. On the other hand, PGP [187] functions with completely decentralized keys, by having users *sign each other's keys*—this is the same mechanism used in the attestation system described in Section 2.11. When Alice gets "Bob's" public key, she checks its signatures to see if *someone she trusts* has signed that key, or some short chain of trustable people, etc. If so, then this key must be genuine (or there is a conspiracy afoot amongst their mutual friends); if not, then the key may be a forgery. This practice of signing the keys of those you vouch for is called the *PGP web of trust* and is the primary safeguard against forged keys. Yenta, for example, uses this technique in *signing attestations* as part of its reputation system.

## 3.4 Structure of the solutions

This section presents solutions to some likely security problems in our architecture, using some of the technology mentioned previously. It presents a *range* of solutions; not every user in every application might want the overhead of the most complete protection, and the elements, while often solving separate problems, sometimes also act synergistically to improve the situation. Finally, for brevity, it omits some details present in the complete design.

### *3.4.1 The nature of identity*

**Uniqueness and confidentiality.** It should not be possible to easily spoof the identity of an agent. For this reason, every agent sports a unique *cryptographic identity*—a *digital pseudonym*. This identity corresponds, essentially, to the *key fingerprint* [187] of the individual agent's public key—a short (128 bits) cryptographic hash of the entire key. In Yenta, this identity is referred to as the user's *Yenta-ID* or *YID*, and is effectively a random number—knowing it does not tell anyone anything about whose *real-life* identity it is. In order to keep some interloper from stealing, say, agent A's pseudonym, any agent communicating with A encrypts messages using A's public key. A can prove that its pseudonym is genuine by being able to decrypt; further, such communications are *interlocked* [155] and have an internal sequence number—itself encrypted—both of which help prevent replay attacks by a man in the middle. Further, of course, such encryption prevents an eavesdropper from intercepting the actual conversation. Thus, even though the actual identity of the user is not known, the user's pseudonym cannot be appropriated.

*Pseudonymity and anonymity are cornerstones of the design*

*The fact that users are by default pseudonymous, and often completely anonymous, is a critical aspect of the security of the architecture.* Consider, for example, what would happen if characteristics that were offered during clustering (Section 2.8.2) automatically identified the user identity that went along with them—the user would be *exposed*, in the terminology of Section 3.2.2. Instead, given the *plausible deniability* feature described in Section 2.8.2, it is at least possible that any given characteristic does not correspond to the agent offering it, meaning that the user is *probably*, and at least *possibly, innocent.* In addition, if third-party subcomparisons and random-reforwarding via cluster broadcasts, also as described in Section 2.8.2, are in use, the user may well be *beyond suspicion.*

The completely decentralized nature of our architecture complicates key distribution. The model adopted is the decentralized model used by PGP [187]. By not relying on a central registry, we eliminate that particular class of failures. And interestingly, the architecture partially eliminates the disadvantage of PGP's decentralized key distribution—that of guaranteeing that any particular public key really does correspond to the individual for which it is claimed. In PGP, we care strongly about actual individuals, but in our architecture, and in the sample application, only the cryptographic ID's are important—for example, Yenta tries to *hide* the true identity of its users unless they arrange to be known to each other.

**Spamming and spoofing.** Unfortunately, this pseudonymity comes at a price: For example, if we are about to be introduced to some user, how can we have any idea who we might be about to be introduced to? Can we know that the last 10 agents we've seen do not all surreptitiously belong the same individual? Can a given user of Yenta, for instance, know that this person won't spam us with junk mail once he discovers our interest in a particular topic? And so forth.

We solve this problem with the *attestation system*, described in Section 2.11. This system provides a set of reputations, which are useful in verifying, if not the identity of a given user, at least whether he or she can make statements about himself or herself that other users will vouch for.

### *3.4.2 Eavesdropping*

The generally-encrypted nature of inter-agent communication makes most eavesdropping, including some but not all man-in-the-middle attacks, quite difficult. However, traffic analysis is still a possibility—for example, if an interloper knows what one Yenta is interested in, watching who it clusters with could be useful.

Fortunately, we have a solution to this, in the broadcasting paradigm mentioned in Section 2.10. In addition, we can use the techniques in Section 3.4.3 to provide additional security.

### *3.4.3 Malicious agents*

If some malicious person was running a subverted version of an agent, what could he discover? The most important information consists of the identities of other agents in the cluster cache—especially if those identities can be those of real users, e.g., their *real names*, and not digital pseudonyms—and the contents in the rumor cache—especially if, again, such text can be correlated to real people. There are therefore two general strategies to combat this: hiding real identifying information as well as possible, and minimizing the amount of text stored in the rumor cache. We shall mention two of the simplest approaches below; other approaches to both problems, involving Mixmaster-style random-reforwarding, secret-sharing protocols, or diffusing pieces of characteristics out to large numbers of third parties for comparison, are possible but are more complicated than necessary for this discussion.

*Hiding identities*

Since users are pseudonymous by default, hiding their identities in large part centers around avoiding traffic analysis. Using the *broadcasting* strategies presented above suffices. For a more complete description, please see Section 2.10.

*Mixing in other agents' data*

A simple technique for protecting users' characteristics against possibly malicious agents is to *mix in other agents' data* when engaging in the comparison and referral process. For a more complete description of this process, please see Section 2.8.3.

*A range of privacy is available*

Depending on which of the strategies above are chosen, and the nature of the characteristics handled by the application, it may be possible to arrange several degrees of user privacy. Using the terminology of Section 3.2.2, these could plausibly range from *possible innocence* to *beyond suspicion*.

### *3.4.4 Protecting the distribution*

There is a final piece of the puzzle—how do users of an agent know that their copy is trustworthy? The easiest approach, of course, is to cryptographically sign the binaries, such that any given binary may be checked for tampering with the authoritative distribution point. But what if the program itself, at the distribution point, had a trojan horse inserted into its *source*, either by the implementors themselves, or by a malicious third party who penetrates the development machine? Even though the source is freely distributed, and may be recompiled by end-users and checked against the binary, what individual user would want to read the *entire* source to check for malicious inclusions? This is, of course, a problem for *any* software, and not just agents in the architecture we present here—but applications such as Yenta are particularly difficult for a user to verify solely from their behavior. After all, they read sensitive files *and* engage in a lot of network traffic—and even worse, the traffic is encrypted, so one cannot even check up on it with a packet sniffer.

In general, those who distribute software have chosen one of three models:

- *Trust us.* Often used by those who do not provide source code at all.
- *Go ahead—read every line of the source code yourself.* This is an infeasible task for almost any reasonable application, and a huge burden.
- *Hope you hear something one way or the other on the net or in the press*. This, too, is both infeasible, error-prone, and subject to a variety of false positives and false negatives.

*The Yenta code vetter*

There is another way. To demonstrate this, we have developed *Yvette*, a Web-based tool which allows multiple people to *collaboratively* evaluate an agent's source code—in this case, Yenta's. A summary of Yvette's capabilities is presented below, and examples of its use are presented in Figure 4 and Figure 5.

Evaluators store cryptographically-signed—hence traceable and non-spoofable—comments on particular pieces of the source where others can view them. The signature covers both the comment *and* the exact text of the code being commented upon. Each individual need only check a small piece of the whole, yet anyone can examine the collected comments and decide whether their contents and coverage add up to an evaluation one can trust.

Yvette presents an interface, via the Web, which allows anyone to ask questions such as:

- Who has commented on this particular piece of code? Are the comments mostly favorable, or not? What is the exact text of the comment(s)?
- What regions have the most or least number of comments associated with them?

Yvette users may also take actions such as:

- Download, for inspection and comment, a piece of the source, which can be a region of lines in a file, a subroutine, a set of subroutines, a set of files, or an entire directory tree.
- Upload cryptographically-signed comments about some piece of downloaded source code.

Note that, since it distributes code that may include cryptographic routines whose export from the US and Canada is illegal [50][87], Yvette must also be aware of which sections of code are sensitive and must use address-based heuristics and questions of the user—only for those parts of Yenta which are cryptographic—to ensure that EAR/ITAR's export restrictions [50][87] are not violated. The heuristics used are the same as those used to control the export of PGP [187], which, while easy to circumvent, are informally viewed as sufficient by at least some of the relevant players in the US government [104].

Using Yvette, therefore, users who wish to help verify a distribution can bite off a small piece of the problem, asking the Yvette server for which pieces of source code have not yet been extensively vetted, perusing other people's comments, and so forth. Users with no programming experience, but who nonetheless wish to check the distribution, may look at everyone else's comments to assure themselves of the integrity of the product.

Yvette thus attempts to encourage a *whole-system* approach to security, in which not only are the agents themselves secure, but their users—who are also part of the system—may easily *trust* the agents' security and integrity. It is hoped that mechanisms such as Yvette will become more popular in software distribution in general, and that it encourages thinking about more than just protocols and cryptography—if we expect widespread adoption of sophisticated agents, the sociology of how users can use and trust them matters, too.

## 3.5 Selected additional topics

There are a few loose ends in the architecture we present here that have not been adequately addressed by the discussion so far. This section attempts to tie them up.

*Central servers*

Let us consider first the central servers that exist in the design, namely the *bootserver* (described in Section 2.7) and the *statserver* (described in Section 2.13). Both of these servers are safe, in the sense of the unlinkability of users' personal data and their actual identities, but in slightly different ways.

The bootserver knows IP addresses. Because of this, it could potentially lead an attacker directly back to an individual. However, the bootserver knows nothing else—in particular, it knows nothing about any user's characteristics, save that the given user runs the application at all.

The statserver, on the other hand, potentially knows quite a bit about all users—in Yenta, for example, it knows information such as how many clusters the user is in, how they tend to use the user interface, what machine architecture Yenta is being run on, and so forth. (Note that it still does not know the detailed contents of individual characteristics, because such information could compromise a user's privacy if revealed, *and* it is unlikely to be so useful for analysis of Yenta's behavior that the risk is worthwhile.) However, the statserver does *not* know user identities or IP addresses at all. Once the data has been stored on disk, the agent's identity and IP address are gone. The only data that the statserver has preserved is a unique *random number* which can be used to differentiate one agent from another, but nothing else.

*Encrypted connections*

As for the safety of the data getting to the statserver in the first place, or between any given pair of agents, note that we have specified that *all* communications are routinely encrypted. The only exception is in data which contains no personal user data, namely bootstrap requests and replies, either via broadcast or to and from the bootserver. For details of how Yenta performs such encryption, see Section 4.8.1.

*Persistent state*

Getting the data between agents is only part of the story, however; we must also consider the storage of the agent's persistent state across shutdowns. In most applications, this is likely to stored be in a filesystem on a disk. If the agent handles personal information, this storage point is a tempting target for an attacker. Furthermore, it is likely that the application may store users' private keys—perhaps the basis of their identity—in this file as well, meaning that an attacker who can read the file can not only violate the user's privacy, but impersonate him or her to other users as well, with potentially serious implications.

It is clear, therefore, that such data should be protected. Exactly how this is to be accomplished is application- and implementation-specific; how Yenta does so is described in Section 4.8.2. Note in particular that this is a rich source of possible security problems, for several reasons:

- A network connection is necessarily a moving target—if an eavesdropper fails to intercept the relevant packets, the attack fails. On the other hand, data stored on disk is vulnerable to compromise from the moment it is created until long afterwards—perhaps even after it is thought deleted, and, to a sophisticated adversary, even after the disk has been formatted [73]. Keeping backups around forever, and failing to adequately encrypt their contents or control physical access to them, only makes this worse.

- If the data is stored encrypted, we have the often-difficult problem of how to securely ask the user for the decryption key. Many environments provide no known-secure method of eliciting such data; in particular, UNIX users who use the X Window System [152] or Telnet [136] are particularly vulnerable to simple packet sniffing, and this is an extremely popular attack. While it is possible for knowledgeable users to use SSH [158] or Kerberos' ktelnet [127], there is often no way for the application to ensure this—and the consequences of a single instance of carelessness could lead to the user's privacy being unknowingly compromised forever afterwards.
- Encrypting the data with the same encryption key every time it is written to disk exposes it to a number of attacks if the data varies [155].

*Random numbers*

Finally, note that we have at many points mentioned the term *random number*—whether explicitly, in Section 2.13's discussion of the ID's generated for statserver, or implicitly, whenever we talk about generating any sort of key—session keys, public/private key pairs, and so forth. We assume here that such random numbers are really *pseudorandom*, e.g., derived from deterministic software.

Where are these random numbers coming from? Certainly not from typical application libraries; most random number sources provided with most operating systems are extremely poor when employed for cryptographic applications. Further, several high-profile examples of poor decisions in sources of random numbers have come to light, such as an early Netscape attempt at SSL [63] which derived its "random" numbers from easily-predictable values provided by the host operating system—this meant that a browser's supposedly-secure, 128-bit-key connection to a server could be broken in around 25 seconds [67].

Thus, the implementor must take great care in selection and use of random numbers in the application. This is common sense in cryptographic circles, but it bears repeating here. Exactly where to find a good source of randomness is always implementation-specific; some operating systems make available random numbers which are derived from turbulent processes (such as disk head performance), but many do not.

For an illustrative example of how Yenta acquires, manages, and uses random numbers, see Section 4.8.3.

## 3.6 Summary

In this chapter, we have described the threat model—what sorts of attacks we consider within the scope of this research. We then presented some background on modern cryptography and how it can help address many of the threats presented; we also discussed how decentralization of the architecture contributes greatly to the protection we afford. Finally, we presented a new method which makes collaboratively evaluating the source code of a critical application easier, and tied up some loose ends.